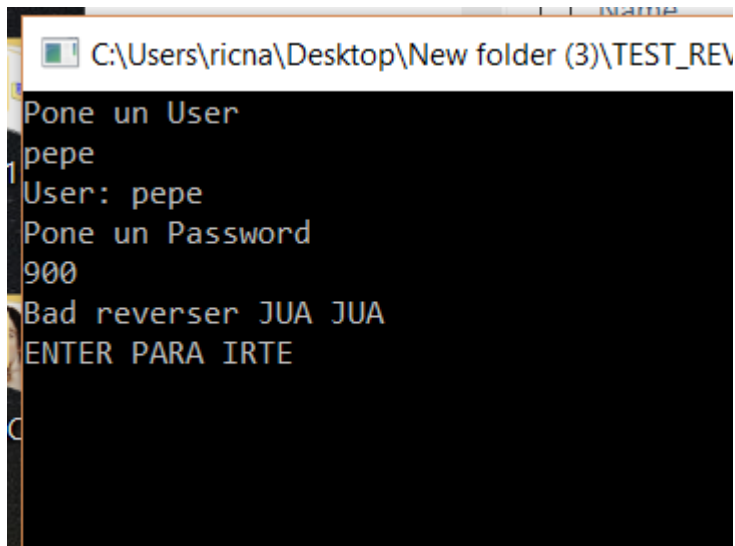


Введение в реверсинг с нуля используя IDA PRO. Часть 12.

Хорошо, чтобы не было скучно, мы будем объединять теорию и некоторые упражнения, в этом случае, это другая программа скомпилированная мной, которая называется **TEST_REVERSER.EXE** и которая очень простая, но она поможет нам увидеть некоторые новые вещи в статическом реверсинге и которые мы проверим в отладчике.

Если мы запустим программу вне **IDA**, то мы увидим.



```
C:\Users\ricna\Desktop\New folder (3)\TEST_REVERSER.EXE
Pone un User
pepe
User: pepe
Pone un Password
900
Bad reverser JUA JUA
ENTER PARA IRTE
```

Нас просят ввести имя пользователя и затем пароль, а потом программа говорит нам, что мы проиграли, и она смеётся над нами.

Давайте откроем её в **IDA**, чтобы увидеть программу в статическом виде.

Поскольку я не использую символы, всё оказывается слегка уродливым.

```

0040142E
0040142E
0040142E ; Attributes: library function
0040142E
0040142E public start
0040142E start proc near
0040142E
0040142E ; FUNCTION CHUNK AT 004012BF SIZE 0000012C BYTES
0040142E ; FUNCTION CHUNK AT 00401428 SIZE 00000006 BYTES
0040142E
0040142E call sub_4017C3
00401433 jmp loc_4012BF
00401433 start endp ; sp-analysis failed
00401433

```

0000082E 0040142E: start (Synchronized with Hex View-1)

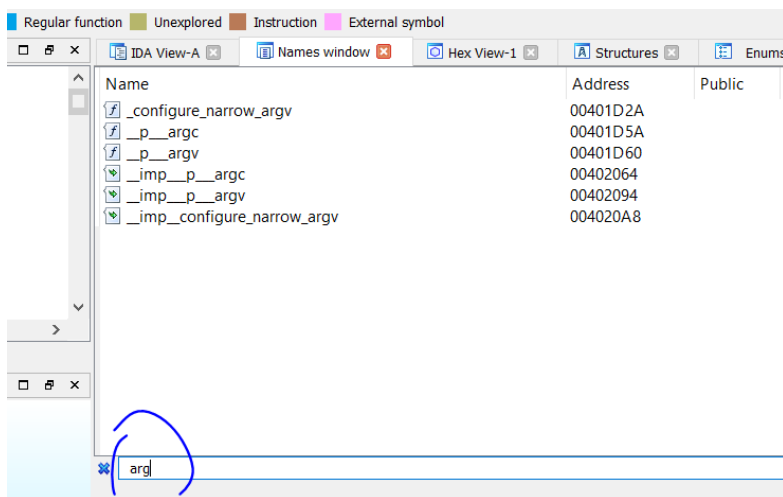
Очевидно, она не открывается в функции **MAIN**, зато открывается в **EP**, но хорошо, действительность почти всегда такова и мы разберемся с этой проблемой.

Один из способов, который мы уже видели, чтобы попасть в `горячую часть программы` - найти строки, мы уже знаем как это делать, также в этих консольных программа на **C++**, это один из способов найти **MAIN**, который почти всегда работает.

Мы знаем, что в функцию передаются аргументы **ARGC**, **ARGV** и т.д. Это аргументы консоли.

INT MAIN(INT ARGC, CHAR *ARGV[])

Мы уже видели в предыдущем примере, что даже если аргументы не используются, они всё равно **ПОМЕЩАЮТСЯ** в стек, иногда это действует по умолчанию для консольных исполняемых файлов, поэтому мы можем искать их во вкладке **NAMES**, чтобы увидеть, есть ли они там или нет.



Отсюда и далее, когда я говорю - "На вкладке **XXX**", Вы уже должны знать, что это открывается в меню **VIEW → OPEN SUBVIEW → XXX**, чтобы не повторять это более.

Здесь с помощью комбинации **CTRL + F** мы фильтруем ввод, введём например **ARG** и мы увидим записи, давайте сделаем двойной щелчок по **_P_ARGC**.

```
00401D5A
00401D5A
00401D5A ; Attributes: thunk
00401D5A
00401D5A __p__argc proc near
00401D5A jmp     ds: __imp__p__argc
00401D5A __p__argc endp
00401D5A
```

Ища ссылки с помощью **X** мы находим

```
0040139A pop     ecx
```

```
0040139B
0040139B loc_40139B:
0040139B call   __p__argv
004013A0 mov    edi, eax
004013A2 call   __p__argc
004013A7 mov    esi, eax
004013A9 call   get_initial_narrow_environment
004013AF push  eax
004013AF push  dword ptr [edi]
004013B1 push  dword ptr [esi]
004013B3 call  sub_401070
004013B8 add    esp, 0Ch
004013BB mov    esi, eax
004013BD call  sub_4019EF
004013C2 test   al, al
004013C4 jnz   short loc_4013CC
```

079A 0040139A: start-94 (Synchronized with Hex View-1)

Здесь мы видим как программа вызывает функции **_P_ARGV** и **_P_ARGC** и то, что она передает содержимое результата в функцию **MAIN**, адрес которой в этом случае - **0x401070**.

Если я посмотрю ту же самую функцию в моей **IDA**, в версии с символами.

```
00401070 ; Attributes: bp-based frame
00401070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401070 main proc near
00401070
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
14,135) 00000470 00401070: main (Synchronized with Hex View-1)
```

И мы видим ссылку.

```
0040139B
0040139B loc_40139B:
0040139B call    __p_argv
004013A0 mov     edi, eax
004013A2 call    __p_argc
004013A7 mov     esi, eax
004013A9 call    __get_initial_narrow_environment
004013AE push   eax ; envp
004013AF push   dword ptr [edi] ; argv
004013B1 push   dword ptr [esi] ; argc
004013B3 call    main
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    __scrt_is_managed_app
004013C2 test   al, al
004013C4 jnz    short loc_4013CC
```

Очевидно это не хорошая идея использовать такие читы, так можно только проверить, что метод для нахождения **MAIN** работает и здесь - это полностью верно, ища ссылки аргументов, которые передаются консолю, мы прибываем к **MAIN**.

Давайте переименуем этот вызов в **MAIN**.

```
0401393 push   dword ptr [esi]
0401395 call   _register_thread_local_exe_atexit_callback
040139A pop    ecx
```

```
0040139B
0040139B loc_40139B:
0040139B call    __p_argv
004013A0 mov     edi, eax
004013A2 call    __p_argc
004013A7 mov     esi, eax
004013A9 call    __get_initial_narrow_envi
004013AE push   eax
004013AF push   dword ptr [edi]
004013B1 push   dword ptr [esi]
004013B3 call    sub_401070
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test   al, al
004013C4 jnz    short loc_4013CC
```

07B3 004013B3: start-7B (Synchronized with

Rename address
Address: 0x401070
Name: main
Maximum length of new names: 15
Local name prefix: @@
 Local name
 Include in names list
 Public name
 Autogenerated name
 Weak name
 Create name anyway
OK Cancel Help

IDA автоматически переименовывает **АРГУМЕНТЫ** узнав, что вышеупомянутая функция – это **MAIN**.

```
00401393 push    dword ptr [esi]
00401395 call    _register_thread_local_exe_atexit_callback
0040139A pop     ecx

0040139B loc_40139B:
0040139B call    _p__argv
004013A0 mov     edi, eax
004013A2 call    _p__argc
004013A7 mov     esi, eax
004013A9 call    get_initial_narrow_environment
004013AE push    eax
004013AF push    dword ptr [edi]; argv
004013B1 push    dword ptr [esi]; argc
004013B3 call    main
004013B8 add     esp, 0Ch
004013BB mov     esi, eax
004013BD call    sub_4019EF
004013C2 test    al, al
004013C4 jnz    short loc_4013CC

000007B3 004013B3: start-7B (Synchronized with Hex View-1)
```

Сейчас, это больше похоже на версию с символами.

```
00401070
00401070
00401070 ; Attributes: bp-based frame
00401070
00401070 ; int __cdecl main(int argc, const char **argv, const char **envp)
00401070 main proc near
00401070
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0

3,247) 00000470 00401070: main (Synchronized with Hex View-1)
```

Мы видим в этом случае, что переменные и аргументы более нумерованы чем в предыдущем примере.

Если мы сделаем двойной щелчок на любой переменной или аргументу, мы увидим статическое представление стека.

```

IDA VIEW-A  Stack of main  Names window  Hex
-00000013      db ? ; undefined
-00000012      db ? ; undefined
-00000011      db ? ; undefined
-00000010      db ? ; undefined
-0000000F      db ? ; undefined
-0000000E      db ? ; undefined
-0000000D      db ? ; undefined
-0000000C      db ? ; undefined
-0000000B      db ? ; undefined
-0000000A      db ? ; undefined
-00000009      db ? ; undefined
-00000008      db ? ; undefined
-00000007      db ? ; undefined
-00000006      db ? ; undefined
-00000005      db ? ; undefined
-00000004  var_4      dd ?
+00000000      s          db 4 dup(?)
+00000004      r          db 4 dup(?)
+00000008  argc         dd ?
+0000000C  argv         dd ?           ; offset
+00000010  envp         dd ?           ; offset
+00000014
+00000014 ; end of stack variables

```

Мы смотрим снизу вверх и видим, что логически первыми идут аргументы функции, которые будут всегда под адресом возврата **R**, так как они передаются через **PUSH** и они сохраняются в стек перед вызовом функции с помощью **CALL**, которая затем сохраняет адрес возврата в стек.

Затем у нас идёт **S** или что, то же самое, что и **STORED EBP**, который является **EBP** функции, которая вызывается функцией **MAIN**, он сохраняется в стек, когда начинает выполняться функция с помощью инструкции **PUSH EBP**.

```

00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0

```

Затем программа помещает **ESP** в **EBP**, помещая значение в **EBP**, которое оно будет иметь в этой функции, чтобы быть **БАЗОЙ** откуда берутся аргументы, которые ниже **БАЗЫ** и локальные переменные, которые выше неё, и наконец инструкция **SUB ESP, 0x94** сдвигает **ESP** выделяя место для переменных и локальных буферов, поэтому они выше, в этом случае размер для буфера будет равен **0x94**, потому что компилятор вычисляет, размер который ему нужен, чтобы

зарезервировать место для переменных и буферов, в соответствии с тем, как мы запрограммировали нашу функцию.

ESP остается со значением выше этого зарезервированного пространства для локальных переменных и **EBP** указывает на **БАЗУ** или **ГОРИЗОНТ**, который делит стек на локальные переменные, которые выше и **СОХРАНЕННЫЙ EBP, АДРЕС ВОЗВРАТА** и **АРГУМЕНТЫ**, которые ниже него.

```
lar function Unexplored Instruction External symbol
Stack of main IDA View-A Names window Hex View-1 Structures
-00000017 db ? ; undefined
-00000016 db ? ; undefined
-00000015 db ? ; undefined
-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
SP+00000090
```

Вот почему в функциях основанных на **EBP**, как только программа вызывает функцию, она сохраняет с помощью инструкции **PUSH EBP** значение **EBP** функции, которую вызывает программа, затем программа помещает **ESP** в **EBP** и это считается теперь как горизонт, вот почему в статическом представлении стека, **IDA** показывает **00000000** как горизонт, а выше видно знаки минус (-), а ниже знаки плюс (+).

Вот почему **VAR_4** имеет значение - **00000004**, потому что переменная берет **EBP** как **БАЗУ** или как **0** и математическим адресом переменной будет **EBP - 4**.

И ниже **ARGC** будет равен **EBP + 8**, это видно по колонки слева.


```

Stack of main
IDA View-A
Names window
He

-00000014 db ? ; undefined
-00000013 db ? ; undefined
-00000012 db ? ; undefined
-00000011 db ? ; undefined
-00000010 db ? ; undefined
-0000000F db ? ; undefined
-0000000E db ? ; undefined
-0000000D db ? ; undefined
-0000000C db ? ; undefined
-0000000B db ? ; undefined
-0000000A db ? ; undefined
-00000009 db ? ; undefined
-00000008 db ? ; undefined
-00000007 db ? ; undefined
-00000006 db ? ; undefined
-00000005 db ? ; undefined
-00000004 var_4 dd ?
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 argc dd ?
+0000000C argv dd ? ; offset
+00000010 envp dd ? ; offset
+00000014
+00000014 ; end of stack variables

```

Здесь мы видим переменную **BUF**, которая является первой переменной над пустой зоной, мы делаем правый клик и выбираем **ARRAY**.

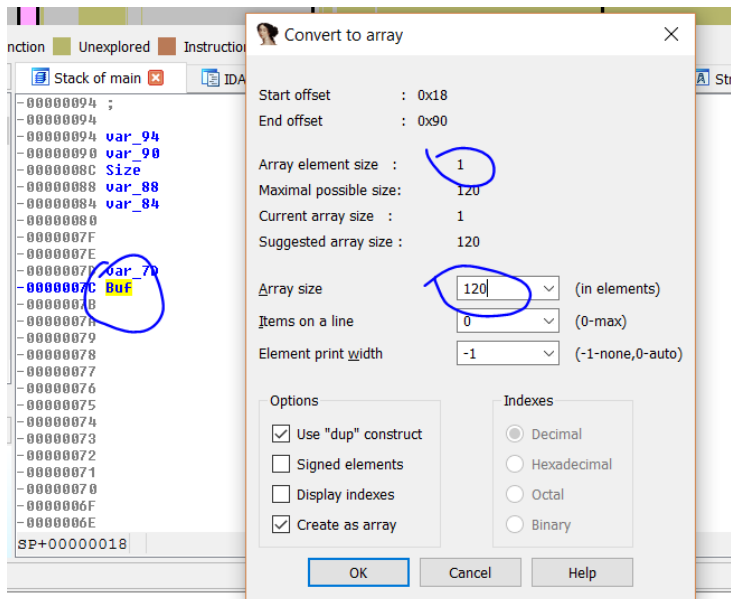
```

Stack of main
IDA View-A
Names window
He

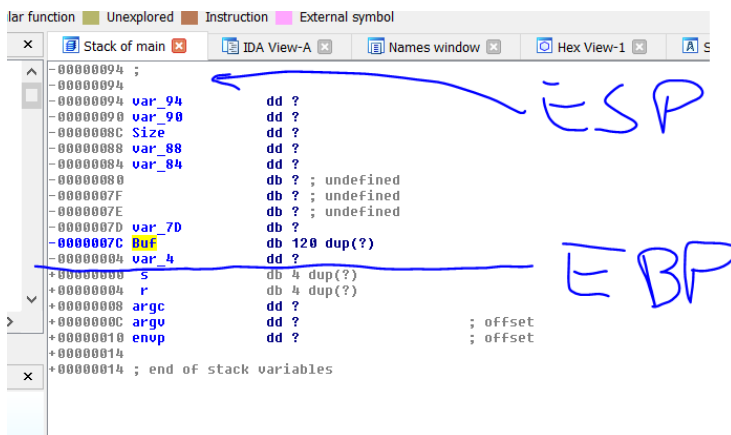
-00000094 ;
-00000094
-00000094 var_94 dd ?
-00000090 var_90 dd ?
-0000008C Size dd ?
-00000088 var_88 dd ?
-00000084 var_84 dd ?
-00000080 db ? ; undefined
-0000007F db ? ; undefined
-0000007E db ? ; undefined
-0000007D var_7D db ?
-0000007C Buf db ?
-0000007B db ? ; undefined
-0000007A db ? ; undefined
-00000079 db ? ; undefined
-00000078 db ? ; undefined
-00000077 db ? ; undefined
-00000076 db ? ; undefined
-00000075 db ? ; undefined
-00000074 db ? ; undefined

```

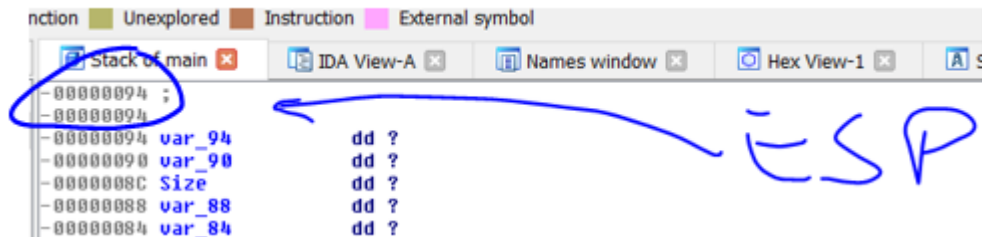
Мы видим, что размер **МАССИВА** равен **120** байт, потому что он состоит из **120** элементов по **1** байту.



Сейчас представление стека стало лучше.



Мы видим базу **EBP** и мы помним, что как только **EBP** и **ESP** равны это равносильно инструкции **MOV EBP, ESP**, затем программа вычитает из **ESP** значение **0x94** и **ESP** теперь начинает работать выше зоны для локальных переменных.



Здесь мы видим область в которой **ESP** будет после инструкции **SUB ESP, 0x94**.

Здесь в левой стороне видно значение **-00000094** или что также равно **ESP = EBP - 094**, очевидно затем значение будет продолжать расти по мере работы программы, между другими подпрограммы и т.д., но всегда пока значение находится внутри этой функции **MAIN** и пока значение не выйдет из этой области, оно будет продолжать работать в области не выше **0x94** зарезервированную часть для переменных, чтобы не наступить на них.

Хорошо, однажды мы уже видели статическое представления стека, давайте отреверсим переменные, поскольку аргументы нам известны (**ARGC, ARGV**, и т.д.)

```

00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0

```

Мы уже видели, что **VAR_4** это переменная **COOKIE_SEGURIDAD** или **CANARY**, мы видим, что инструкция считывает это значение, затем **XORum** его с помощью **EBP** и сохраняет его в стек, чтобы защитить стек от **ПЕРЕПОЛНЕНИЯ**, поэтому давайте переименуем эту переменную.

```

00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 COOKIE_SEGURIDAD= dword ptr -4
00401070 argc= dword ptr 8
00401070 argv= dword ptr 0Ch
00401070 envp= dword ptr 10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
00401098 push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    sub_4011B0
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size]
004010AE push    eax ; Size
004010AF lea    ecx, [ebp+Buf]
004010B2 push    ecx ; Buf

```

153 | (499,242) 00000480 00401080: main+10

Равно как и в предыдущем примере **API** функция **PRINTF** не имеет символов и она не отображается, но я наблюдаю строки для неё, которые она печатает в консоли и мы видим эту функцию по адресу **0x4011B0**.

```

Names window  Hex View-1  Structures  Enums
00401180 arg_0= dword ptr 8
00401180 arg_4= byte ptr 0Ch
00401180
00401180 push ebp
00401181 mov ebp, esp
00401183 sub esp, 8
00401186 call sub_401000
0040118B lea eax, [ebp+arg_4]
0040118E mov [ebp+var_4], eax
004011C1 mov ecx, [ebp+var_4]
004011C4 push ecx
004011C5 push 0
004011C7 mov edx, [ebp+arg_0]
004011CA push edx
004011CB push 1
004011CD call ds:__acrt_iob_func
004011D3 add esp, 4
004011D6 push eax
004011D7 call sub_401040
004011DC add esp, 10h
004011DF mov [ebp+var_8], eax
004011E2 mov [ebp+var_4], 0
004011E9 mov eax, [ebp+var_8]
004011EC mov esp, ebp
004011EE pop ebp
004011EF retn
5D7:004011D7: sub_4011B0+27

```

И здесь внутри по адресу **0x401040** мы видим.

```

Names window  Hex View-1  Structures  Enums
00401040 arg_0= dword ptr 8
00401040 arg_4= dword ptr 0Ch
00401040 arg_8= dword ptr 10h
00401040 arg_C= dword ptr 14h
00401040
00401040 push ebp
00401041 mov ebp, esp
00401043 mov eax, [ebp+arg_C]
00401046 push eax
00401047 mov ecx, [ebp+arg_8]
00401048 push ecx
00401048 mov edx, [ebp+arg_4]
0040104E push edx
0040104F mov eax, [ebp+arg_0]
00401052 push eax
00401053 call sub_401030
00401058 mov ecx, [eax+4]
0040105B push ecx
0040105C pop edx, [eax]
0040105E push edx
0040105F call ds:__stdio_common_vfprintf
00401065 add esp, 10h
00401068 pop ebp
00401069 retn
00401069 sub_401040 endp
00401069
0440:00401040: sub_401040

```

Так что, давайте переименуем функцию по адресу **0x4011B0** в функцию с именем **PRINTF**.

```

00401070
00401070 push ebp
00401071 mov ebp, esp
00401073 sub esp, 94h
00401079 mov eax, __security_cookie
0040107E xor eax, ebp
00401080 mov [ebp+COOKIE_SEGURIDAD], eax
00401083 mov [ebp+var_7D], 0
00401087 mov [ebp+var_88], 0
00401091 mov [ebp+Size], 8
0040109B push offset aPoneUnUser ; "Pone un User\n"
004010A0 call printf
004010A5 add esp, 4
004010A9 mov eax, [ebp+Size]
004010AE push eax
004010AF mov [ebp+var_7D], eax

```

Давайте идти дальше.

```

00401070 mov     esp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push   offset aPoneUnUser ; "Pone un User\n"

```

Director	Typ	Address	Text
	w	main+21	mov [ebp+Size], 8
Down	r	main+38	mov eax, [ebp+Size]
Down	r	main+BE	mov edx, [ebp+Size]

OK Cancel Search Help

Мы видим, что размер переменной инициализируется с помощью числа **8** и больше это значение никогда не изменяется, есть только два чтения среди следующих ссылок, поэтому мы переименуем размер этой переменной в имя **_CONST_8**.

```

00401070 enop= dword ptr 10h
00401070
00401070 push   ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, __security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size_CONST_8], 8
0040109B push   offset aPoneUnUser ; "Pone un User\n"
004010A0 call   printf
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size_CONST_8]
004010AE push   eax ; Size
004010AF lea    ecx, [ebp+Buf]
004010B2 push   ecx ; Buf
004010B3 call   ds:gets_s
004010B9 add     esp, 8

```

Затем мы видим вызов функции **GETS_S**, которая является эволюцией функции **GETS**, но с ограничением по количеству вводимых символов, которые мы можем ввести(это такая защита), в этом случае максимальным значением будет **8**, которое помещается в **EAX** и передается как аргумент с помощью **PUSH EAX**, а затем **LEA** получает адрес переменной **BUF** или **BUFFER**.

gets_s, _getws_s

Visual Studio 2015 | Other Versions ▾

Gets a line from the stdin stream. These versions of gets_s, _getws_s have security enhancements described in [Security Features in the CRT](#).

Syntax

```
char *gets_s(  
    char *buffer,  
    size_t sizeInCharacters  
);
```

Конечно, если мы введём больше символов чем **8** и нажмём **ENTER**, функция также будет обрезать ввод и случится возврат.

Так что мы знаем, что в **BUF** будет помещаться имя **ПОЛЬЗОВАТЕЛЯ**, которое мы набрали и что оно будет иметь максимум **8** символов.

```
004010B3 call    ds:gets_s  
004010B9 add     esp, 8  
004010BC lea    edx, [ebp+Buf]  
004010BF push   edx ; Str  
004010C0 call   strlen  
004010C5 add     esp, 4  
004010C8 mov    [ebp+var_90], eax  
004010CE mov    [ebp+var_84], 0  
004010D8 jmp    short loc_4010E9
```

Здесь мы видим, что потом программа передаёт с помощью **PUSH EDX** адрес буфера снова, как аргумент к **API** функции **STRLEN**, чтобы получить длину строки, которая сейчас находится в **BUF** и соответствует введённому **ПОЛЬЗОВАТЕЛЮ**, и сохраняет длину в переменной **VAR_90** через регистр-результат **EAX**, так что мы переименовываем **VAR_90** в **LEN_USER**.

```
004010AE push   eax ; Size  
004010AF lea   ecx, [ebp+Buf]  
004010B2 push  ecx ; Buf  
004010B3 call  ds:gets_s  
004010B9 add   esp, 8  
004010BC lea   edx, [ebp+Buf]  
004010BF push  edx ; Str  
004010C0 call  strlen  
004010C5 add   esp, 4  
004010C8 mov   [ebp+len_USER], eax  
004010CE mov   [ebp+var_84], 0  
004010D8 imn  short loc_4010E9
```

```

004010BF push    edx                ; Str
004010C0 call   strlen
004010C5 add    esp, 4
004010C8 mov    [ebp+len_USER], eax
004010CE mov    [ebp+var_84], 0
004010D8 jmp    short loc_4010E9

004010E9 loc_4010E9:
004010E9 mov    ecx, [ebp+var_84]
004010EF cmp    ecx, [ebp+len_USER]
004010F5 jge    short loc_401110

004010F7 mov    edx, [ebp+var_84]
004010FD movsx  eax, [ebp+edx+Buf]
00401102 add    eax, [ebp+var_88]
00401108 mov    [ebp+var_88], eax
0040110E jmp    short loc_4010DA

```

Синяя стрелка всегда указывает на переход назад, который может быть **ЦИКЛОМ**, по адресу **0x4010CE** программа инициализирует счетчик **LOOP VAR_84**, мы также видим, что по адресу **0x4010F5** находится условный переход, который оценивает условие выхода из цикла, счетчик начинается с **НУЛЯ**, и он будет увеличиваться в каждом цикле, и программа выйдет из цикла, когда счетчик будет больше или равен длине, которую мы ввели в **LEN_USER**.

```

004010BF push    edx                ; Str
004010C0 call   strlen
004010C5 add    esp, 4
004010C8 mov    [ebp+len_USER], eax
004010CE mov    [ebp+CONTAD00R], 0
004010D8 jmp    short loc_4010E9

004010E9 loc_4010E9:
004010E9 mov    ecx, [ebp+CONTAD00R]
004010EF cmp    ecx, [ebp+len_USER]
004010F5 jge    short loc_401110

004010F7 mov    edx, [ebp+CONTAD00R]
004010FD movsx  eax, [ebp+edx+Buf]
00401102 add    eax, [ebp+var_88]
00401108 mov    [ebp+var_88], eax
0040110E jmp    short loc_4010DA

```

Счетчик увеличивается к концу **ЦИКЛА** здесь.

```

RTEV" ...
004010DA loc_4010DA:
004010DA mov    eax, [ebp+CONTAD00R]
004010E0 add    eax, 1
004010E3 mov    [ebp+CONTAD00R], eax

```

Здесь он помещает значение **СЧЁТЧИКА** в **EAX** увеличивая его и затем снова сохраняет.

```

004010F7 mov    edx, [ebp+CONTADOOR]
004010FD movsx  eax, [ebp+edx+Buf]
00401102 add    eax, [ebp+var_88]
00401108 mov    [ebp+var_88], eax
0040110E jmp    short loc_4010DA

```

Здесь программа помещает первый байт **БУФЕРА** из **EBP + EDX + BUF** в **EAX**, поскольку **EBP + BUF** складывается со **СЧЕТЧИКОМ**, который сейчас равен нулю, то он будет увеличиваться пока работает **ЦИКЛ**, мы видим, что здесь будет складываться все значения символов, которые я набираю, поэтому переменная **VAR_88** которая начинается с нуля, будет складываться в каждом цикле **HEX** значения каждого символа строки **БУФЕРА**.

Мы видим инструкцию, которую мы до этого даже ещё не видели - **MOVSX**.

MOVSX и **MOVZX**.

Обе инструкции берут байт и помещают его в регистр, в случае с **MOVZX** заполняются с помощью нулей старшие байты, в то время как в случае с **MOVSX** учитывается знак байта, если он положительный или меньше или равен **0x7F** он заполняется с помощью нулей и если он отрицательный или равен **0x80** или больше заполняется с помощью **0xFF**.

MOVZX EAX, [XXXX]

Если содержимое **XXXX** будет равно **0x40**, **EAX** будет равен **0x00000040**.

Также например существует инструкция **MOVZX EAX, CL**.

Этот случай похожий, инструкция будет брать значение байта и заполнит с помощью нулей старшие байты.

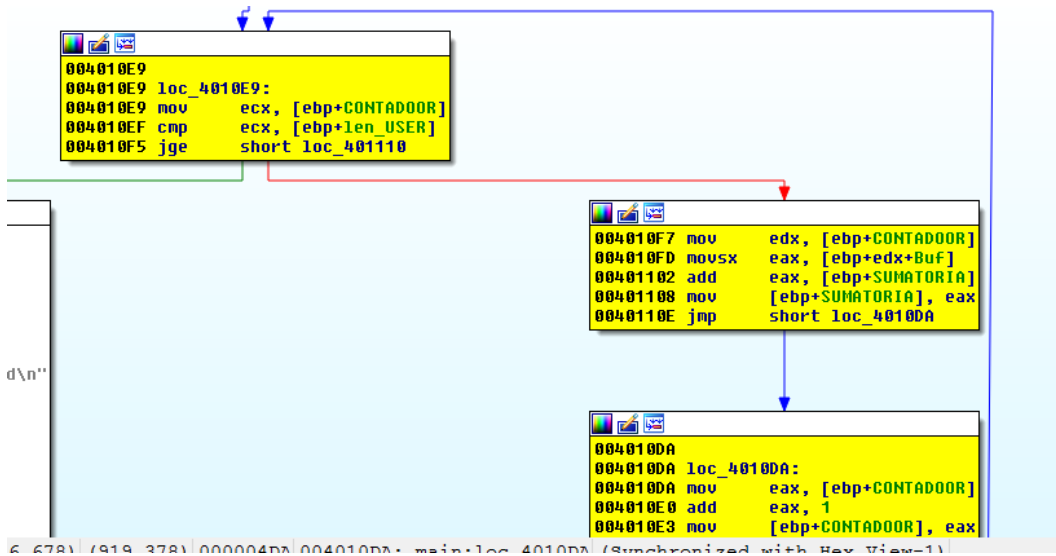
MOVSX EAX, CL

Инструкция принимает во внимание знак байта, если **CL** - например равен **0x40**, **EAX** будет равен **0x00000040** и если бы он был бы равен **0x85**, в этом случае, поскольку у него отрицательный знак и это значение отрицательное **EAX** будет равен **0xFFFFF85**.

Также, так как мы вводим символы букв и чисел через консоль, эти символы являются положительными **HEX** значениями, так что у нас не будет никаких проблем, программа будет складывать значения один за одним и сохранять.

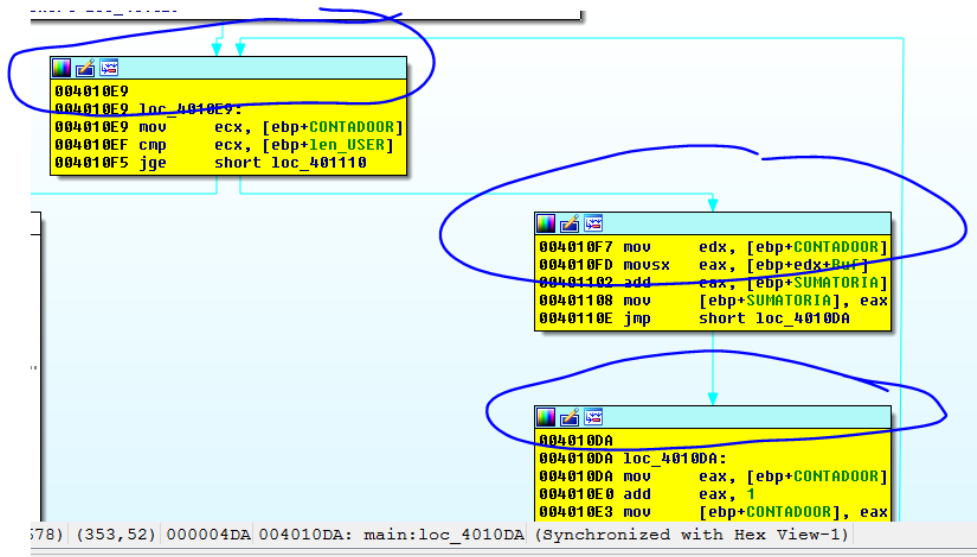

```
004010F7 mov     edx, [ebp+CONTADOOR]
004010FD movsx  eax, [ebp+edx+Buf]
00401102 add     eax, [ebp+SUMATORIA]
00401108 mov     [ebp+SUMATORIA], eax
0040110E jmp     short loc_4010DA
```

Мы видим, что **ЦИКЛ** - это сложение символов, мы покрасим их тем же цветом.

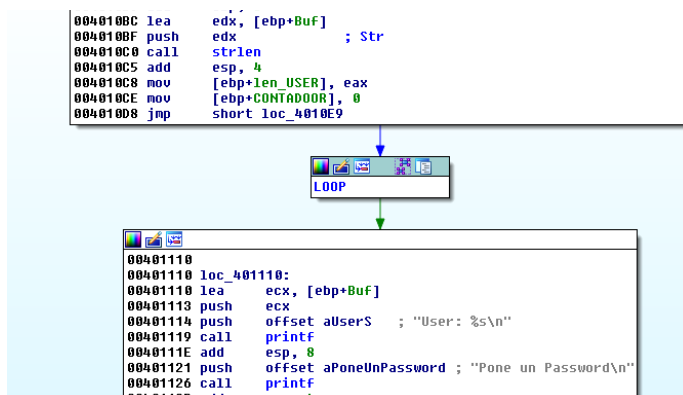
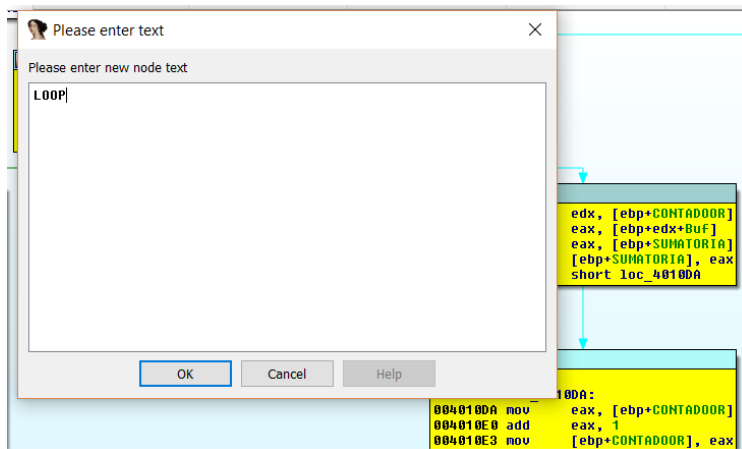


Также я немного увеличил их, перетащив и отпустив нижний блок немного выше.

Есть люди, которые, чтобы убрать блоки с экрана, если они хотят, чтобы они их не беспокоили, группируют их, с помощью нажатия **CTRL** и делая клик в верхнюю панель каждого блока.



Делаем правый клик и выбираем **GROUP NODES**, а затем вводим имя, например **LOOP**.



Последнее, если Вы хотите увидеть блоки, которые спрятаны, то это можно сделать через **UNGROUP NODES**.

Затем программа выводит слово **ПОЛЬЗОВАТЕЛЬ** и говорит, что нужно ввести **ПАРОЛЬ**.

```

00401110 loc_401110:
00401110 lea    ecx, [ebp+Buf]
00401113 push  ecx
00401114 push  offset aUserS ; "User: %s\n"
00401119 call  printf
0040111E add   esp, 8
00401121 push  offset aPoneUnPassword ; "Pone un Password\n"
00401126 call  printf
0040112B add   esp, 4
0040112E mov   edx, [ebp+Size_CONST_8]

```

Затем программа вызовет снова функцию **GET_S** используя тот же самый буфер и тоже максимальное значение вводимых символов.

```

00401121 push  offset aPoneUnPassword ; "Pone un Password\n"
00401126 call  printf
0040112B add   esp, 4
0040112E mov   edx, [ebp+Size_CONST_8]
00401134 push  edx ; Size
00401135 lea  eax, [ebp+Buf]
00401138 push  eax ; Buf
00401139 call  ds:gets_s

```

Программа может повторно использовать тот же **БУФЕР** для **ПАРОЛЯ**, в любом случае программа полностью рассчитала **СУММУ HEX** значений символов **ПОЛЬЗОВАТЕЛЯ** и она больше не будет использовать строку **ПОЛЬЗОВАТЕЛЬ**.

```

0040113F add   esp, 8
00401142 lea  ecx, [ebp+Buf]
00401145 push ecx ; Str
00401146 call ds:atoi
0040114C add   esp, 4
0040114F mov  [ebp+var_94], eax

```

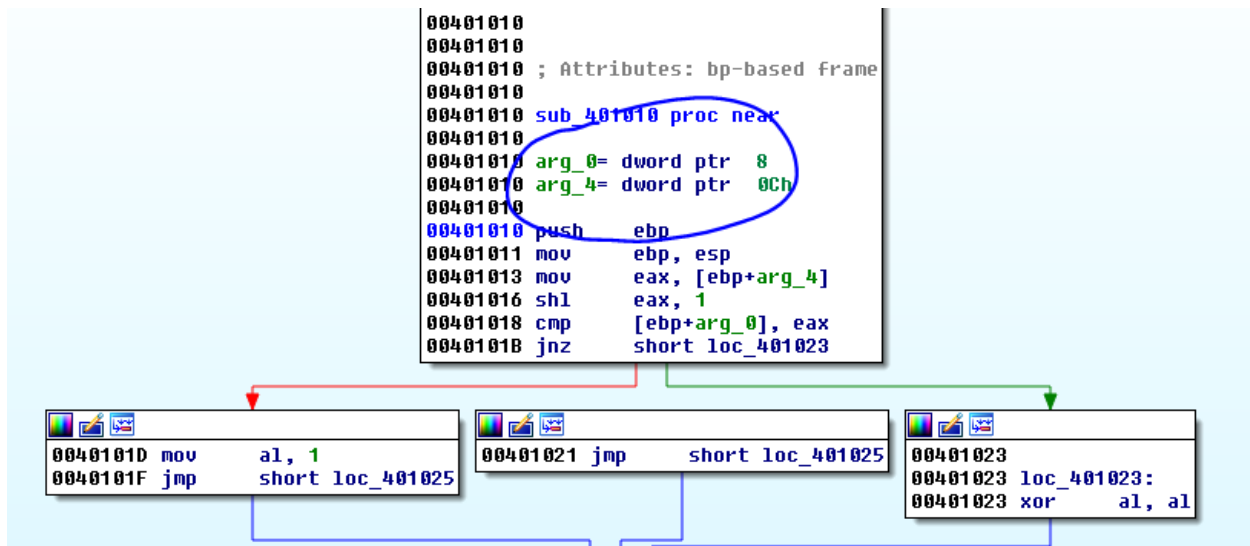
Сейчас она возьмёт **ПАРОЛЬ** и преобразует его в **HEX** как в предыдущем примере используя **atoi**.

```

00401142 lea  ecx, [ebp+Buf]
00401145 push ecx ; Str
00401146 call ds:atoi
0040114C add   esp, 4
0040114F mov  [ebp+VALOR_PASSWORD], eax
00401155 mov  edx, [ebp+VALOR_PASSWORD]
0040115B push edx
0040115C mov  eax, [ebp+SUMATORIA]
00401162 push eax
00401163 call sub_401010
00401168 add   esp, 8

```

Здесь передаётся значение **VALOR_PASSWORD** с помощью инструкции **PUSH EDX** и суммируется с помощью **PUSH EAX**, это будут два аргумента, которые передаются в функцию по адресу **0x401010**, давайте введём их.



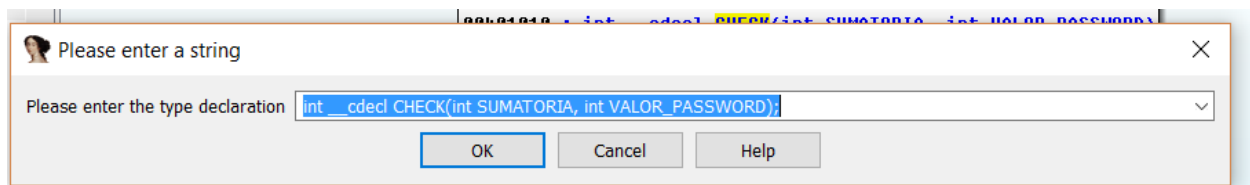
Здесь мы видим два аргумента, очевидно, что тот, который ниже будет **VALOR_PASSWORD** так как передается с помощью **PUSH** в стек первым и второй, который кладется следующим, будет суммой, и он будет выше.

```

00401010 sub_401010 proc near
00401010 SUMATORIA= dword ptr 8
00401010 VALOR_PASSWORD= dword ptr 0Ch
00401010
00401010 push    ebp
00401011 mov     ebp, esp
00401013 mov     eax, [ebp+VALOR_PASSWORD]
00401016 shl    eax, 1
00401018 cmp    [ebp+SUMATORIA], eax
0040101B jnz    short loc_401023

```

Я переименовываю их согласно этому, и затем, чтобы проверить нормально ли всё, сделайте правый щелчок по адресу **SUB_0x401010** и выберите **SET_TYPE**.



При этом **IDA** попытается объявить функцию со своими аргументами, чтобы показать их в ссылке и мы переименуем также это в функцию **CHECK**.

```

00401010
00401010
00401010 ; Attributes: bp-based frame
00401010
00401010 ; int cdecl CHECK(int VALOR_PASSWORD, int SUMATORIA)
00401010 CHECK proc near
00401010
00401010 VALOR_PASSWORD= dword ptr 8
00401010 SUMATORIA= dword ptr 0Ch
00401010
00401010 push ebp
00401011 mov ebp, esp
00401013 mov eax, [ebp+SUMATORIA]
00401016 shl eax, 1
00401018 cmp [ebp+VALOR_PASSWORD], eax
0040101B jnz short loc_401023

```

И если мы пойдём к ссылке.

```

00401107 call ds:getc_3
0040113F add esp, 8
00401142 lea ecx, [ebp+Buf]
00401145 push ecx ; Str
00401146 call ds:atoi
0040114C add esp, 4
0040114F mov [ebp+VALOR_PASSWORD], eax
00401155 mov edx, [ebp+VALOR_PASSWORD]
00401158 push edx ; VALOR_PASSWORD
0040115C mov eax, [ebp+SUMATORIA]
00401162 push eax ; SUMATORIA
00401163 call CHECK
00401168 add esp, 8
0040116B mov [ebp+var_7D], al
0040116E movzx ecx, [ebp+var_7D]
00401172 test ecx, ecx

```

Мы видим, что **IDA** распространит имена и говорит мне, что у **EAX** есть **СЛОЖЕНИЕ** и **EDX** - это **VALOR_PASSWORD**.

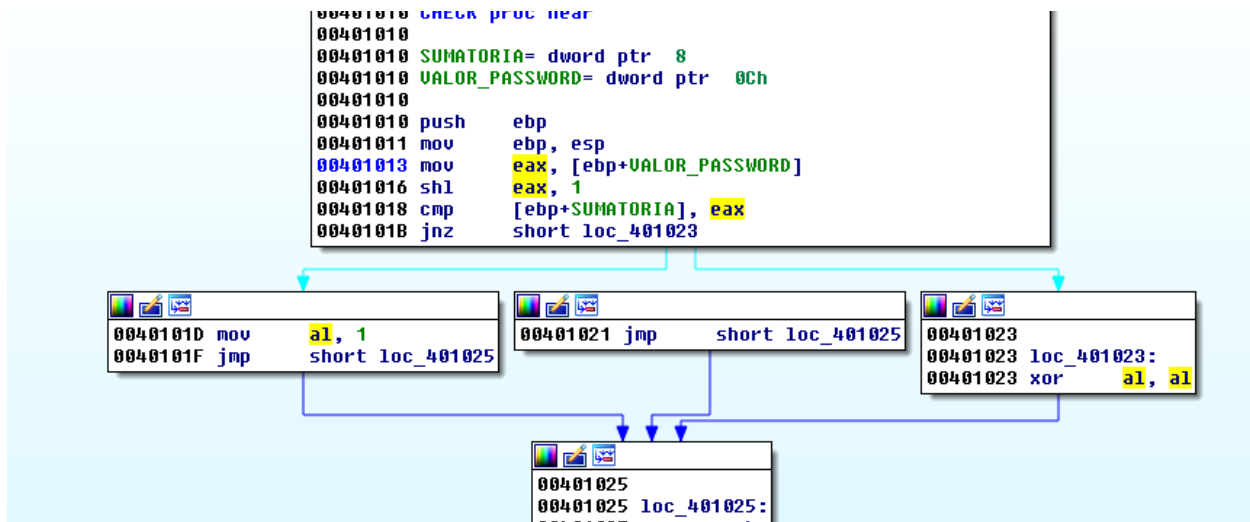
И что делает функция **CHECK** с этими двумя аргументами?

Мы видим, что она сравнивает их, но сначала она берет значение **PASSWORD** и делает с ним операцию **SHL EAX, 1**

Examples

shl eax, 1 — Multiply the value of EAX by 2 (if the most significant bit is 0)

Мы знаем, что **SHL** сдвигает биты влево, заполняя нулями те, которые исчезают на другой стороне, но в частности **SHL REG, 1** - это равносильно умножению на **2**.



Программа берет значение пароля, умножает его на **2** и сравнивает его с суммой символов слова **ПОЛЬЗОВАТЕЛЬ**.

```
Python>ord("p")
112
```

При этом мы получаем числовое значение символа, мы можем сделать формулу, которая суммирует все символы строки **pepe**, которую я использую как **ПОЛЬЗОВАТЕЛЬ**.

```
Python>hex(ord("p")+ord("e")+ord("p")+ord("e"))
0x1aa
```

Мы видим, что сумма равна **0x1AA**, с другой стороны значение, которые мы ввели как пароль будет умножено на **2** перед сравнением с этим **0x1AA**, поэтому правильный пароль должен быть значением, которое при умножении на два даёт нам **0x1AA**.

$$X * 2 = 0x1AA$$

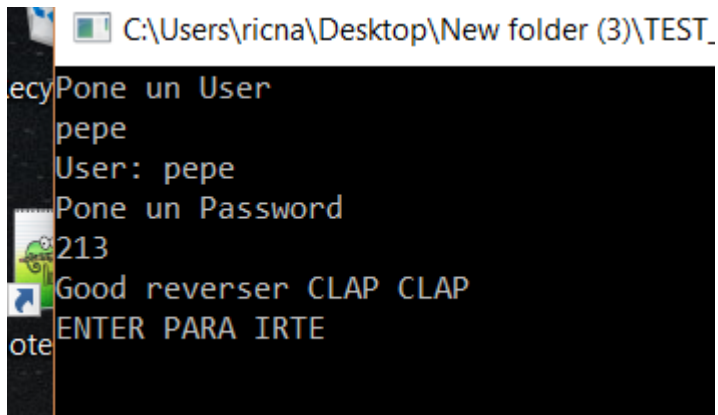
Давайте проясним, что эта программа имеет ограничения, если сложение даёт нечетное число, то невозможно, чтобы существовало значение **X**, которое при умножении на **2** даёт нам результат нечетное число, поэтому эти имена пользователей не имеют решений, в этой программе имеют решения только имена пользователей, чья сумма при сложении чётная.

Мы очищаем строку и вводим.

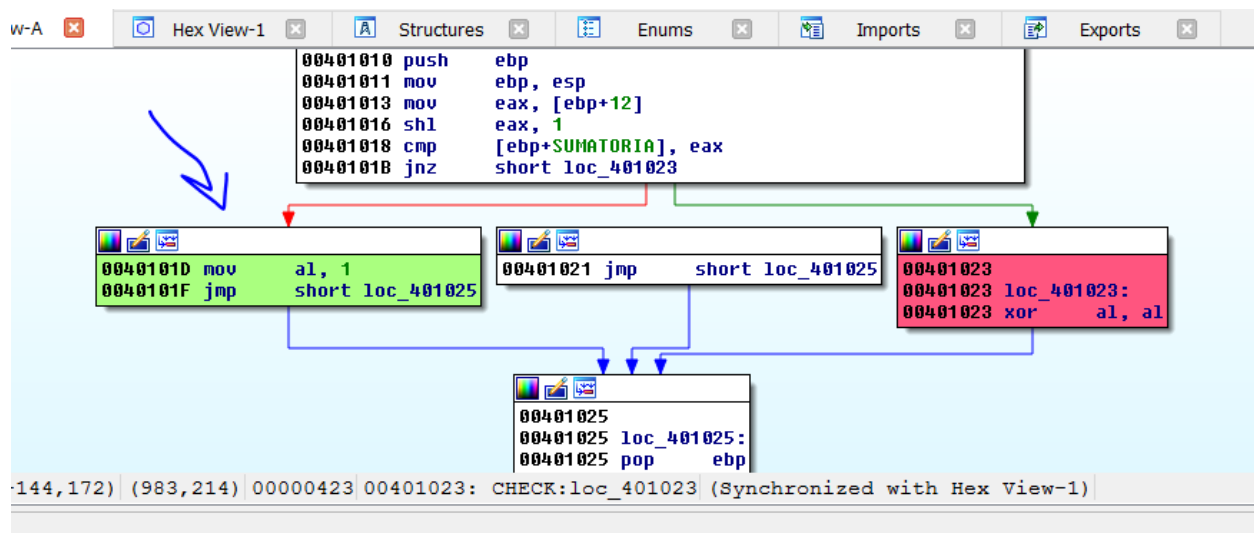
$X = 0x1AA / 2$ и ответ поступает в десятичной системе счисления, очевидно, что с помощью **ATOI** он переводится из десятичной системы счисления в **HEX**.

```
Python>0x1aa/2
213
```

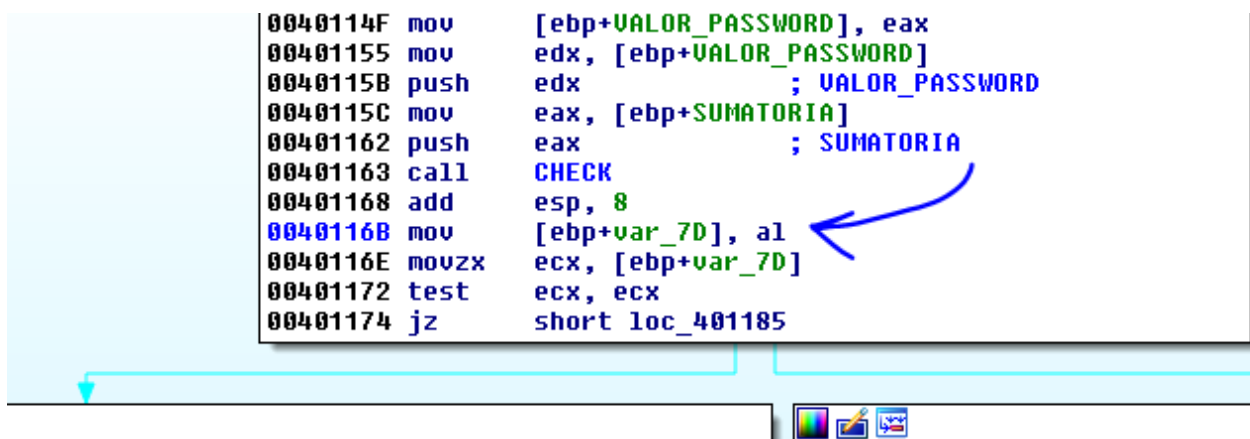
Если я введу **ИМЯ ПОЛЬЗОВАТЕЛЯ** как **pepe**, а **ПАРОЛЬ** как **213**, что произойдёт?



Конечно, я вижу, когда сравнение одинаково внутри функции проверки.



Если они не равны, программа идёт в красный блок и возвращает **НУЛЬ**, а если они равны программа идёт в зелёный блок и возвращает **ЕДИНИЦУ**, давайте посмотрим, что произойдёт с этим возвращаемым значением.



Программа сохраняет его здесь, давайте переименуем это значение в **FLAG_EXITO**.

```

00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx ; VALOR_PASSWORD
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax ; SUMATORIA
00401163 call   CHECK
00401168 add     esp, 8
0040116B mov     [ebp+FLAG_EXIT0], al
0040116E movzx   ecx, [ebp+FLAG_EXIT0]
00401172 test    ecx, ecx
00401174 jz     short loc_401185

ffset aGoodReverserC1 ; "Good reverser CLAP CLAP\nENTER PARA IRT"...
printf
sp, 4
short loc_401192

00401185
00401185 loc_401185: ; "Bad reverser JUA
00401185 push    offset aBadReverserJua
0040118A call   printf
0040118F add     esp, 4

```

100.00% (149,1115) (960,202) 0000058A 0040118A: main+11A (Synchronized with Hex View-1)

Так что, поскольку, если мы видим **0**, то тогда пойдём к **BAD REVERSER**, а если **AL** равно **1**, то идём к **GOOD BOY** как это и случается у нас здесь.

Мне бы очень понравилось, если бы вы отладили этот крэкми и проверили всё, что мы вместе реверсили, установив **BPs** и увидев значения в каждом случае до окончательной проверки.

До встречи в **13-мой** части.



Перевод на английский: **IvinsonCLS**

Перевод на русский с испанского+английского: **Яша_Добрый_Хакер(Ростовский фанат Нарвахи).**

Перевод специально для форума системного и низкоуровневого программирования - **WASM.IN**

02.09.2017

Версия 1.0

******* Всем, кому интересна судьба этого и следующих проектов, прошу посетить ресурс **yasha.su** *******

Этот, последующие и предыдущие **PDF** можно будет в дальнейшем взять на этом ресурсе.

